Análise de Complexidade
(continuação)

Sergio Canuto
sergio.canuto@ifg.edu.br

Relembrando… Função de custo f(x) "Exemplo - Registros de um Arquivo"

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recuperar registros do arquivo.
- O problema: dada uma chave qualquer, localize o registro que contenha esta chave.
- O algoritmo de pesquisa mais simples é o que faz a **pesquisa seqüencial**.
- Seja *f* uma função de complexidade tal que $f(n)$ é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - **melhor caso: $f(n) = 1$ (registro procurado é o primeiro consultado);**
  - **pior caso: $f(n) = n$ (registro procurado é o último consultado ou não está presente no arquivo);**
  - **caso médio: $f(n) = (n + 1)/2$.**

Relembrando… Função de custo f(x) “Exemplo - Registros de um Arquivo”

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.
- Se p_i for a probabilidade de que o *i*-ésimo registro seja procurado, e considerando que para recuperar o *i*-ésimo registro são necessárias *i* comparações, então

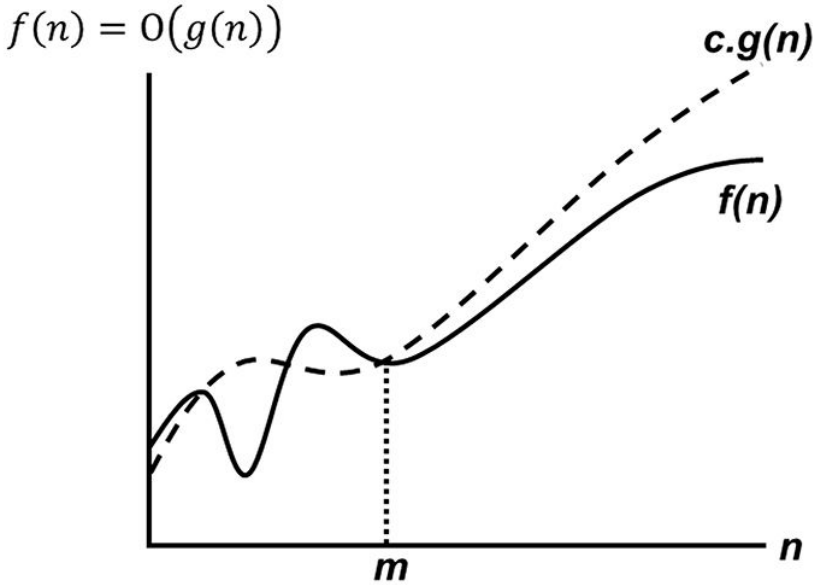
  $f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$.
- Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n$, $0 \leq i < n$.
- Neste caso:

$$f(n) = \frac{1}{n}(1+2+3+\cdots+n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}.$$

- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

Relembrando… Comportamento Assintótico - notação *O*

- Matematicamente, a notação *O* é assim definida: uma função custo $f(n)$ é $O(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $f(n) \leq cg(n)$.
- Em outras palavras, para todos os valores de **n** à direita do valor **m**, o resultado da nossa função custo $f(n)$ é sempre **menor ou igual** ao valor da função usada na notação *O*, $g(n)$, multiplicada por uma constante **c**.



Comportamento Assintótico - notação *O*

- **Exemplo**: $f(n) = (n + 1)^2$.
  - Logo $f(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$.
  - Isso porque $(n + 1)^2 \leq 4n^2$ para $n \geq 1$.
- **Exemplo**: $f(n) = n$.
  - Sabemos que $f(n)$ é $O(n^2)$, pois para $n \geq 0$, $n \leq n^2$.
- **Exemplo:** $f(n) = n^2$
  - $f(n)$ não é $O(n)$.
  - Suponha que existam constantes c e m tais que para todo $n \geq m$, $\boldsymbol{n^2 \leq cn}$. (ou n.n $\leq cn$)
  - Logo $c \geq n$ para qualquer $n \geq m$, e não existe uma constante c que possa ser maior ou igual a n para todo n.

Comportamento Assintótico - notação *O*

- **Exemplo**: $f(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$.
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$.
  - A função $f(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que $f(n)$ é $O(n^3)$.

Comportamento Assintótico - Operações com a notação *O*

$f(n)$	$=$	$O(f(n))$
$c \times O(f(n))$	$=$	$O(f(n))$ $c =$ *constante*
$O(f(n)) + O(f(n))$	$=$	$O(f(n))$
$O(O(f(n))$	$=$	$O(f(n))$
$O(f_1(n)) + O(f_2(n))$	$=$	$O(max(f_1(n), f_2(n)))$
$O(f_1(n))O(f_2(n))$	$=$	$O(f_1(n)f_2(n))$
$f_1(n)O(f_2(n))$	$=$	$O(f_1(n)f_2(n))$

Comportamento Assintótico - Operações com a notação *O*

Exemplo: regra da soma $O(f(n)) + O(f_2(n))$.

- Suponha três trechos cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$.
- O tempo de execução dos dois primeiros trechos é $O(max(n, n^2))$, que é $O(n^2)$.
- O tempo de execução de todos os três trechos é então $O(max(n^2, n \log n))$, que é $O(n^2)$.

Exemplo: O produto de $[\log n + k + O(1/n)]$ por $[n + O(\sqrt{n})]$ é $n \log n + kn + O(\sqrt{n} \log n)$

Questão exemplo - Complexidade

A área de complexidade de algoritmos abrange a medição da eficiência de um algoritmo frente à quantidade de operações realizadas até que ele encontre seu resultado final. A respeito desse contexto, suponha que um arquivo texto contenha o nome de N cidades de determinado estado, que cada nome de cidade esteja separado do seguinte por um caracter especial de fim de linha e classificado em ordem alfabética crescente. **Considere um programa que realize a leitura sequencial linha a linha desse arquivo**, à procura de nome de cidade.

Com base nessa descrição, verifica-se que a complexidade desse programa é:

(A) O(1).
(B) O(N).
(C) O(logN).
(D) $O(N^2)$.
(E) O(N logN).

Questão exemplo - Complexidade

Tendo como referência o algoritmo abaixo, julgue a afirmação seguinte como verdadeira ou falsa: “O algoritmo em apreço é $O(n^3)$”

```
1   inteiro pontuacaoFinal (inteiro n)
2      inteiro i, valor;
3      início
4         valor <- 0;
5         para i de 1 até n faça
6           valor <- valor + i * i * i;
7         fim para
8         retorne valor;
9      fim
```

Questão exemplo - Complexidade

Indique se como verdadeiro ou falso as seguintes afirmações:

a) $f(n)=2n+10$ é $O(n)$
b) $f(n)=(3/2)n(n+1)$ é $O(n)$
c) $f(n)=n/1000$ é $O(n)$
d) $f(n)=2^{n+1}$ é $O(2^n)$

Questão exemplo - Complexidade

Considere dois algoritmos A e B com funções de complexidade de tempo $a(n) = n^2 - n + 500$ e $b(n) = 47n + 47$, respectivamente. Para quais valores de n o algoritmo A leva menos tempo para executar do que B?

Questão exemplo - Complexidade

3) Considere dois algoritmos A e B com funções de complexidade de tempo $a(n) = n^2 - n + 500$ e $b(n) = 47n + 47$, respectivamente. Para quais valores de n o algoritmo A leva menos tempo para executar do que B?

a(n)<b(n)

$n^2 - n + 500 < 47n + 47$

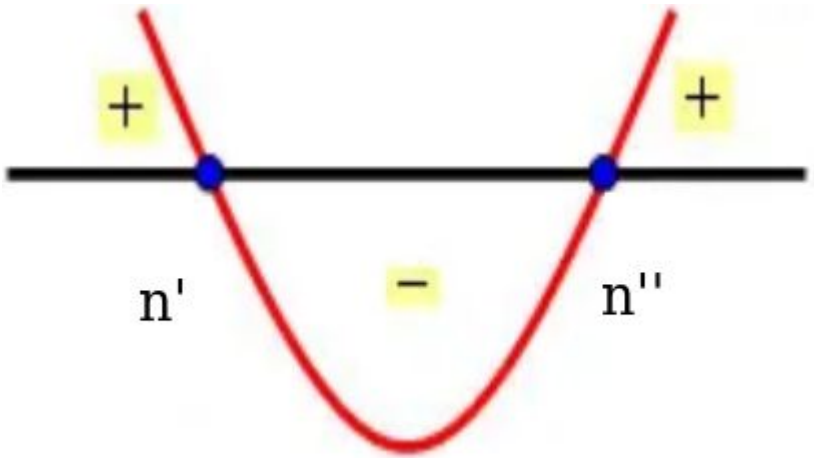
$n^2 - 48n + 453 < 0$

...

n' = 12,909

n'' = 35,091

R.: Entre 13 e 35



Atividade - Complexidade

Considerando a notação *O*, analise o seguinte fragmento de código e sua função de custo.

Gere a função de custo considerando as seguintes **instruções simples** para computar a função de custo do algoritmo:

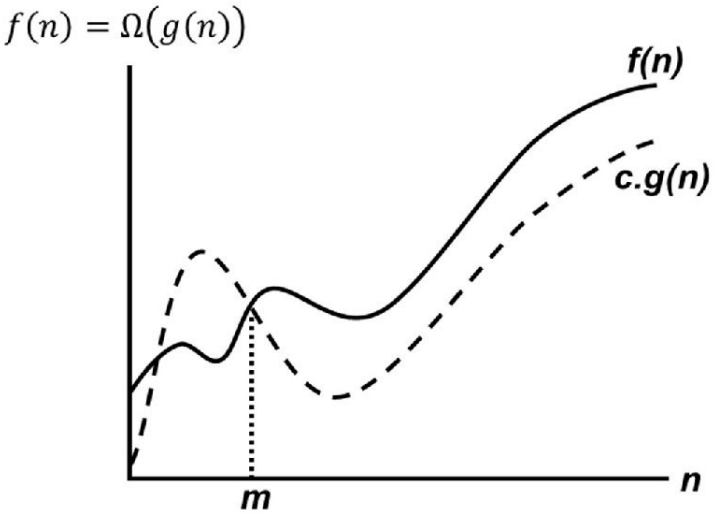
```
1 int i,j,k;
2
3 for(i=0; i < N; i++){
4    for(j=0; j < N; j++){
5       R[i][j] = 0;
6          for(k=0; k < N; k++)
7             R[i][j] += A[i][k] * B[k][j];
8    }
9 }
```

Comportamento Assintótico - Ω (lê-se grande-omega)

- A notação **Grande-O** é a mais utilizada, pois é o caso de mais fácil identificação (limite superior sobre o tempo de execução do algoritmo). Para diversos algoritmos, o pior caso ocorre com frequência.
- A notação Ω (lê-se grande-omega) descreve o **limite assintótico inferior** de um algoritmo. Trata-se de uma notação utilizada para analisar o **melhor caso** do algoritmo.
    - A notação $\Omega(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, maior ou igual a n^2. Em outras palavras, o custo do algoritmo original é, no **mínimo**, tão ruim quanto n^2.

Comportamento Assintótico - notação Ω

- Matematicamente, a notação Ω é assim definida: uma função custo $f(n)$ é $\Omega(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para n ≥ m, temos $f(n) \geq cg(n)$.
- Em outras palavras, para todos os valores de **n** à direita do valor **m**, o resultado da nossa função custo $f(n)$ é sempre **maior ou igual** ao valor da função usada na notação Ω, $g(n)$, multiplicada por uma constante **c**.



Exemplo: **Comportamento Assintótico -** notação Ω

Seja um algoritmo cuja função de custo é 3n+3. Encontre as constantes positivas **c** e **m** que mostram que o algoritmo, no melhor caso, é Ω(n)

Exemplo: **Comportamento Assintótico -** notação Ω

Seja um algoritmo cuja função de custo é 3n+3. Encontre as constantes positivas **c** e **m** que mostram que o algoritmo, no melhor caso, é Ω(n)

Definição: Uma função de custo $f(n)$ é $\Omega(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para n ≥ m, temos $f(n) \geq cg(n)$.

- No nosso caso:
  - f(n)=3n+3
  - g(n)=n

Portanto:

- $f(n) \geq cg(n)$
- 3n+3 ≥ cn.

Vamos escolher c=2. Resolvendo a inequação, 3n+3≥ 2n, sabemos que ela é válida quando n≥-3. Logo, para m=1 (constante positiva) e c=2 atendemos a definição. Portanto existem duas constantes c e m que satisfazem a definição, e podemos dizer que a função de custo 3n+3 é Ω(n)

Exemplo: **Comportamento Assintótico -** notação Ω

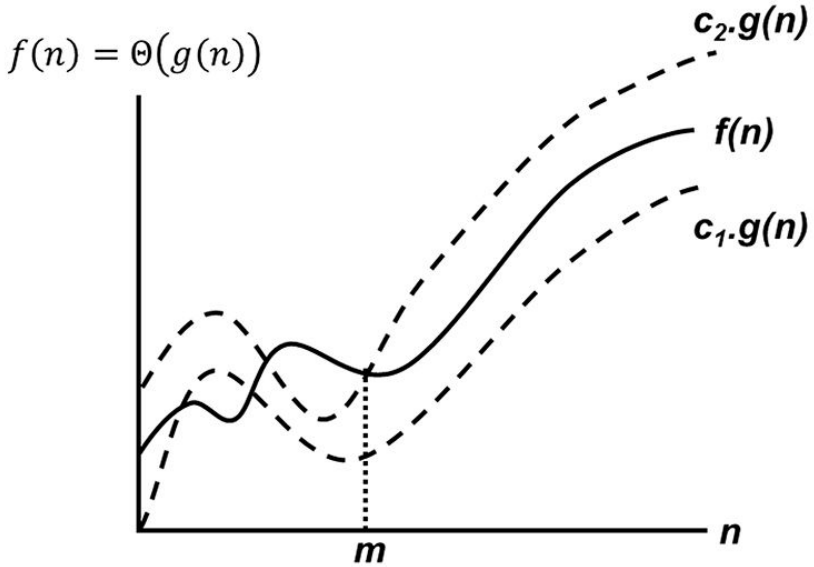
- **Exemplo**: Para mostrar que $f(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer $c = 1$, e então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.
- **Exemplo**: Seja $f(n) = n$ para n ímpar ($n \geq 1$) e $f(n) = n^2/10$ para n par ($n \geq 0$).
  - Neste caso $f(n)$ é $\Omega(n)$, bastando considerar $c = 1$ e $n = 1, 3, 5, 7, \ldots$

Comportamento Assintótico - notação Θ

- A notação Θ (lê-se grande-theta) descreve o **limite assintótico firme** (ou **estreito**) de um algoritmo. Trata-se de uma notação utilizada para analisar o limite inferior e superior do algoritmo.
- A notação $\Theta(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, igual a n^2. Em outras palavras, o custo do algoritmo original é n^2 dentro de um fator constante acima e abaixo.

Comportamento Assintótico - notação Θ

- Matematicamente, a notação Θ é assim definida: uma função custo $f(n)$ é $\Theta(g(n))$ se existem três constantes positivas **c**1, **c**2 e **m**, tais que, para $n \geq m$, temos $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$.
- Em outras palavras, para todos os valores de **n** à direita do valor **m**, o resultado da nossa função custo $f(n)$ é sempre **igual** ao valor da função usada na notação Θ, $g(n)$, quando esta função é multiplicada por constantes **c**1 e **c**2.



Comportamento Assintótico - notação Θ

- Um exemplo simples desse tipo de notação consiste em mostrar que a seguinte função custo do nosso algoritmo:

$$f(n) = \frac{1}{2}n^2 - 3n$$

é $\Theta(n^2)$. Para tanto, iremos definir constantes positivas $\mathbf{c}_1$, $\mathbf{c}_2$ e $\mathbf{m}$, tais que:

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

para todo $n \geq m$. Dividindo por n^2 temos:

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Comportamento Assintótico - notação Θ

- Conseguimos definir as constantes c_1, c_2 e n que satisfazem a inequação abaixo?

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- Sim! escolhendo $c_1 >= 1/14$, $c_2 >= 1/2$ e n >=7 (m=7)
- Outras constantes podem existir, mas o importante é que existe alguma escolha para as três constantes.
- Logo, podemos dizer que:

$$f(n) = \frac{1}{2}n^2 - 3n \quad \text{é} \quad \Theta(n^2).$$

As notações o (pequeno o) e ω (pequeno ômega)

- o e ω são muito parecidas com as notações O e Ω, respectivamente
- Enquanto as notações O e Ω possuem uma relação de **menor ou igual** e **maior ou igual**, as notações o e ω possuem uma relação de **menor** e **maior**.
  - A notação $o(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, sempre menor do que n^2. Matematicamente, uma função custo $f(n)$ é $o(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $\boldsymbol{f(n) < cg(n).}$
  - A notação $\omega(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, sempre maior do que n^2. Matematicamente, uma função custo $f(n)$ é $\omega(g(n))$ se existem duas constantes positivas **c** e **m**, tais que, para $n \geq m$, temos $\boldsymbol{f(n) > cg(n).}$

Análise de Complexidade: Comparação de Programas

- Se *f* é uma **função de complexidade** para um algoritmo *F* , então $O(f)$ é considerada a **complexidade assintótica** ou o comportamento assintótico do algoritmo *F* .
- A relação de dominação assintótica permite comparar funções de complexidade.
- Entretanto, se as funções *f* e *g* dominam assintoticamente uma a outra, então os algoritmos associados são equivalentes.
- Nestes casos, o comportamento assintótico não serve para comparar os algoritmos.
- Por exemplo, considere dois algoritmos *F* e *G* aplicados à mesma classe de problemas, sendo que *F* leva três vezes o tempo de *G* ao serem executados, isto é, $f(n) = 3g(n)$, sendo que $O(f(n)) = O(g(n))$.
- Logo, o comportamento assintótico não serve para comparar os algoritmos *F* e *G*, porque eles diferem apenas por uma constante.

Análise de Complexidade: Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.
- Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$.
- Porém, as constantes de proporcionalidade podem alterar esta consideração.
- Exemplo: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?
  - depende do tamanho do problema.
  - Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possui tempo $100n$.
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$.
  - Entretanto, quando n cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$.

Análise de Complexidade: Classes de PROBLEMAS

- $f(n) = O(1)$.
  - Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**.
  - Uso do algoritmo independe de n.
  - As instruções do algoritmo são executadas um número fixo de vezes.
- $f(n) = O(\log n)$.
  - Um algoritmo de complexidade $O(\log n)$ é dito de **complexidade logarítmica**.
  - Típico em algoritmos que transformam um problema em outros menores.
  - Pode-se considerar o tempo de execução como menor que uma constante grande.
  - Quando n é mil, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$.
  - Para dobrar o valor de $\log n$ temos de considerar o quadrado de n.
  - A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.

Análise de Complexidade: Classes de PROBLEMAS

- $f(n) = O(n)$.
  - Um algoritmo de complexidade $O(n)$ é dito de **complexidade linear**.
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - É a melhor situação possível para um algoritmo que tem de processar/produzir n elementos de entrada/saída.
  - Cada vez que n dobra de tamanho, o tempo de execução também dobra.

- $f(n) = O(n \log n)$.
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e juntando as soluções depois.
  - Quando n é 1 milhão, $n\log_2 n$ é cerca de 20 milhões.
  - Quando n é 2 milhões, $n\log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro.

Análise de Complexidade: Classes de PROBLEMAS

- $f(n) = O(n^2)$.
  - Um algoritmo de complexidade $O(n^2)$ é dito de **complexidade quadrática**.
  - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.
  - Quando n é mil, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução é multiplicado por 4.
  - Úteis para resolver problemas de tamanhos relativamente pequenos.
- $f(n) = O(n^3)$.
  - Um algoritmo de complexidade $O(n^3)$ é dito de **complexidade cúbica**.
  - Úteis apenas para resolver pequenos problemas.
  - Quando n é 100, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução fica multiplicado por 8.

Análise de Complexidade: Classes de PROBLEMAS

- $f(n) = O(2^n)$.
  - Um algoritmo de complexidade $O(2^n)$ é dito de **complexidade exponencial**.
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.
  - Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado.

- $f(n) = O(n!)$.
  - Um algoritmo de complexidade $O(n!)$ é dito de complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa **força bruta** para na solução do problema.
  - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.
  - $n = 40 \rightarrow$ um número com 48 dígitos.

.

Análise de Complexidade: Classes de Problemas

A seguir, são apresentadas algumas classes de complexidade de problemas comumente usadas:

- $O(1)$: **ordem constante**. As instruções são executadas um número fixo de vezes. Não depende do tamanho dos dados de entrada.
- $O(logN)$: **ordem logarítmica**. Típica de algoritmos que resolvem um problema transformando-o em problemas menores.
- $O(N)$: **ordem linear**. Em geral, certa quantidade de operações é realizada sobre cada um dos elementos de entrada.
- $O(NlogN)$: **ordem log linear**. Típica de algoritmos que trabalham com particionamento dos dados. Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos.
- $O(N_2)$: **ordem quadrática**. Normalmente, ocorre quando os dados são processados aos pares. Uma característica deste tipo de algoritmos é a presença de um aninhamento de dois comandos de repetição.
- $O(N_3)$: **ordem cúbica**. É caracterizado pela presença de três estruturas de repetição aninhadas.
- $O(2_N)$: **ordem exponencial**. Geralmente, ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático.
- $O(N!)$: **ordem fatorial**. Geralmente, ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático. Possui um comportamento muito pior que o exponencial.

Análise de Complexidade: Classes de Problemas

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Análise de Complexidade: Classes de Problemas

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Recorrência & Complexidade

- Quando um algoritmo contém uma chamadas recursivas, seu tempo de execução pode freqüentemente ser descrito por uma recorrência;
- Com o ferramental aprendido até o momento, não somos capazes de analisar a complexidade de algoritmos recursivos;
- Para os algoritmos recursivos, a ferramenta principal desta análise não é uma somatória, mas um tipo especial de equação chamada relação de recorrência.
- Uma recorrência é uma equação ou desigualdade que descreve uma função recursiva em termos de seu valor em entradas menores;
    - Uma função é chamada de *função recursiva* quando chama a si mesma durante a sua execução.

Recorrência

- Para cada procedimento recursivo é associada uma função de complexidade T(n) desconhecida, onde n mede o tamanho dos argumentos para o procedimento.

- Equação de recorrência: maneira de definir uma função por uma expressão envolvendo a mesma função.

Relação de Recorrência - Exemplo Fatorial

- Questão: Como analisamos a complexidade de uma função fatorial?
- Contexto:
  - Como calcular o fatorial de 4 (definido como 4!)?
    - Para calcular o fatorial de 4, multiplica-se o número 4 pelo fatorial de 3 (definido como 3!). O procedimento termina quando atingimos zero (0!=1).
  - Em outras palavras:
    - Caso base está associado ao término: 0!
    - A recorrência está associada ao procedimento 4!=4*3!=4*3*2!...
      - Generalizando o processo temos que o fatorial de **N** é igual a **N** multiplicado pelo fatorial de **(N - 1)**, ou seja, $N! = N * (N - 1)!$.

Relações de Recorrência - Exemplo Fatorial

- Como fazer a análise de recorrência do algoritmo que calcula o fatorial de um número?

Função recursiva para cálculo do fatorial

```
01  int fatorial (int n){
02    if (n == 0)
03      return 1;
04    else
05      return n * fatorial(n-1);
06  }
```

Função recursiva para cálculo do fatorial

```
01  int fatorial (int n){
02    if (n == 0)
03      return 1;
04    else
05      return n * fatorial(n-1);
06  }
```

Relações de Recorrência - Fatorial

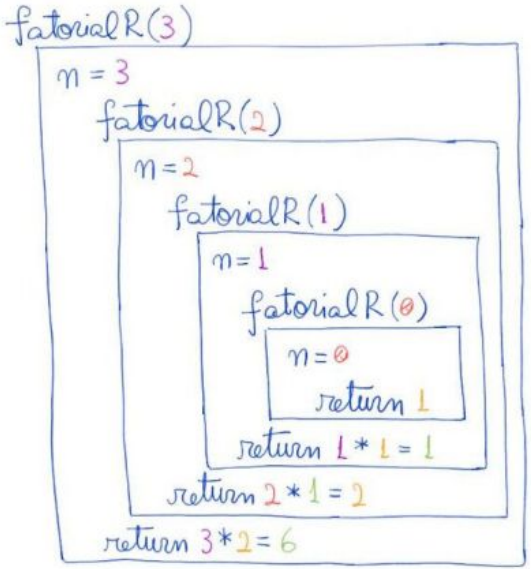
- Qual a relação de recorrência?
  - $T(n)=T(n-1)+1$

Função recursiva para cálculo do fatorial

```
01  int fatorial (int n){
02    if (n == 0)
03      return 1;
04    else
05      return n * fatorial(n-1);
06  }
```

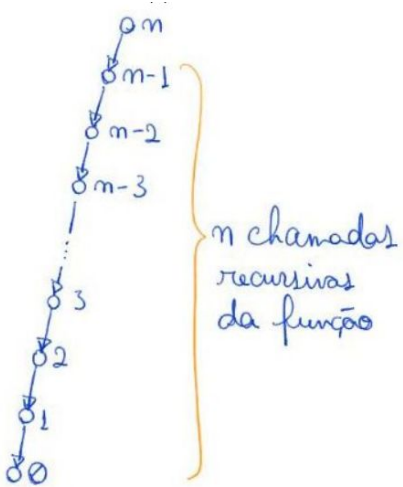
Relações de Recorrência - Fatorial

- Qual a relação de recorrência?
  - A definição de uma função recursiva é, uma expressão que descreve uma função em termos de entradas menores da função. No caso do fatorial:



Relações de Recorrência - Fatorial

- Qual a relação de recorrência?
  - A definição de uma função recursiva é, uma expressão que descreve uma função em termos de entradas menores da função. No caso do fatorial:
  - Podemos avaliar as chamadas recursivas pela **representação da árvore de recursão**
    - n chamadas, cada chamada realiza uma operação -- O(n)



Relações de Recorrência

- Muitos algoritmos se baseiam em recorrência.
  - Ferramenta importante para a solução de problemas combinatórios.
    - Usualmente, não utilizam estruturas de repetição, apenas comandos condicionais, atribuições etc., podemos erroneamente imaginar que essas funções possuem complexidade $O(1)$.
    - Na verdade, para saber a complexidade de um algoritmo recursivo precisamos resolver a sua relação de recorrência.
    - Queremos uma **fórmula fechada** que nos dê o valor da função em termos de seu parâmetro n.
      - No caso anterior
        - T(n)=T(n-1)+1 (**caso n>1**)
        - T(n)=1 **(caso base, n=1)**

Relação de Recorrência - busca

- Exemplo de recorrência:
  - Considere o algoritmo “pouco formal” abaixo:
  - O algoritmo inspeciona n elementos de um conjunto qualquer;
  - De alguma forma, isso permite descartar 2/3 dos elementos e fazer uma chamada recursiva sobre um terço do conjunto original.

```
Algoritmo Pesquisa(vetor)
   if vetor.size() ≤ 1 then
     inspecione elemento;
   else
      inspecione cada elemento recebido (vetor);
      Pesquisa(vetor.subLista(1, (vetor.size() / 3) );
   end if
end.
```

Relação de Recorrência - Busca

- Montando a equação de recorrência:

```
L1: Algoritmo Pesquisa(vetor)
L2:    if  vetor.size() ≤ 1 then
L3:      inspecione  elemento;
L4:    else
L5:       inspecione cada elemento recebido (vetor);
L6:       Pesquisa(vetor.subLista(1, (vetor.size() / 3) );
L7:    end if
L8: end.
```

- Caso base da recursão:
  - O custo da linha 2 é O(1)
  - O custo da linha 3 é O(1)

Relação de Recorrência - Busca

- Montando a equação de recorrência:

```
L1: Algoritmo Pesquisa(vetor)
L2:   if vetor.size() ≤ 1 then
L3:     inspecione elemento;
L4:   else
L5:     inspecione cada elemento recebido (vetor);
L6:     Pesquisa(vetor.subLista(1, (vetor.size() / 3));
L7:   end if
L8: end.
```

- Monte a equação de recorrência...

Relação de Recorrência - Busca

- Montando a equação de recorrência:

$$T(n) = \begin{cases} 1, \ se\, n \leq 1 \\ T(n/3) + n, \ caso\, contrário \end{cases}$$

- Resolva a equação de recorrência...

Relação de Recorrência - Busca

- Resolvendo a equação de recorrência:

$$T(n) = \begin{cases} 1, \ se\, n \leq 1 \\ T(n/3) + n, \ caso\, contrário \end{cases}$$

$$\begin{aligned} T(n) &= n + T(n/3) \\ T(n/3) &= n/3 + T(n/3/3) \\ T(n/3/3) &= n/3/3/3 + T(n/3/3/3) \\ &\dots \\ T(n/3/3\dots/3) &= n/3/3/3\dots/3 + T(n/3/3/3\dots/3) \\ T(1) &= 1 \end{aligned}$$

$$T(n) = n + n/3 + n/3/3 + \dots + n/3/3\dots/3/3 + 1$$

Relação de Recorrência - Busca

$$T(n) = \begin{cases} 1, \; se \;\; n \leq 1 \\ T(n/3) + n, \; caso \;\; contrário \end{cases}$$

$$T(n) = n + n/3 + n/3/3 + \ldots + n/3/3/3\ldots/3 + 1$$

- A formula representa a soma de uma série geométrica de razão 1/3, multiplicada por n, e adicionada de T(n/3/3/3/3 $\cdots$/3), que é menor ou igual a 1.

$$T(n) = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{3}\right)^i + 1$$

Relação de Recorrência - Busca

$$T(n) = \begin{cases} 1, \ se\, n \le 1 \\ T(n/3) + n, \ caso\, contrário \end{cases}$$

$$T(n) = n + n/3 + n/3/3 + ... + n/3/3/3.../3 + 1$$

$$T(n) = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{3}\right)^i + 1 \rightarrow usando: \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$T(n) = n\left(\frac{1}{1-1/3}\right) + 1 = \frac{3n}{2} + 1$$

portanto

$$T(n) \in O(n)$$

Referências

Estrutura de Dados descomplicada em Linguagem C (André Backes): Cap 2;

Projeto de Algoritmos (Nivio Ziviani): Capítulo 1;

Atividades

https://docs.google.com/document/d/1XTmrkkmdXHVz0vVge84tALuy2gjsMgWycVvTQ3sssXg/edit?usp=sharing